PATENT **Docket No. END920040015US1**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

INVENTORS:	**Chua, Sook C.**	Confirmation No. **7732**
APPLICATION NO.	**10/804,644**	
FILED:	**March 19, 2004**	Examiner: **Cloud, J.**
CASE NO.	**END920040015US1**	Group Art Unit: **2444**

TITLE: **J2EE APPLICATION VERSIONING STRATEGY**

FILED ELECTRONICALLY

MAIL STOP AMENDMENT
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF SOOK C. CHUA UNDER 37 C.F.R. §1.131

Sir:

I, Sook C. Chua, hereby declare as follows:

1. I am the inventor of the subject matter claimed in the above-identified patent application.

2. I have been informed that U.S. Patent Application Publication No. 2005/0050548 to Sheinis et al. ("Sheinis") has been cited as prior art by the United States Patent and Trademark Office with respect to the above-identified patent application. I also have been informed that the filing date of Sheinis is August 28, 2003.

3. This declaration is to establish conception and reduction to practice of the invention claimed in claims 1, 2, 4-9, 11-16 and 18-22 of the above-identified application in the United States prior to August 28, 2003.

4. Prior to August 28, 2003, I conceived of and developed (i.e., reduced to practice) a working software program that met all of the limitations of claims 1, 2, 4-9, 11-16 and 18-22 of my patent application, as more particularly detailed below.

5. Prior to August 28, 2003 I created a prototype of the invention and tested it using data from the customer database of an airline, and confirmed that the invention worked.

6. Exhibit A attached hereto is a paper prepared by me after the successful testing of the prototype which discloses details of the prototype and of the invention disclosed and claimed in the above-identified patent application. The date appearing on the original of this document has been redacted from the copy attached hereto. However, the date is prior to August 28, 2003 and accurately reflects a date on which the document existed.

7. In particular, as set forth in claims 1, 8, 15, and 22 of the subject application, a major premise of the claimed invention is the ability to manage the invocation of multiple versions of a J2EE program among multiple clients, in which the multiple versions are stored on a single application server. In claims 1, 8, and 15, it is specified that this is accomplished by interposing a JNDI proxy between each client and the application server, and the JNDI proxy directs the appropriate version to the appropriate client.

8. This invention is described in Exhibit A which, as noted above, was written prior to August 28, 2003 and after a prototype of the invention was made and found to work. Specifically, on pages 4 – 8, there is described and shown (particularly in Figures 2, 4, and 5) the method and configuration whereby multiple versions of a J2EE prgpram are stored on a single application server, and each client has a JNDI proxy situated between it and the application server. It is further described how this method/configuration enables each client to invoke a different version of the J2EE program using these proxies.

9. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment or both, under Section 1001 of Title 18 of the United States Code, and that such willful statements may jeopardize the validity of the application or any patent issued thereon.

8/13/2010
Dated

Sook C. Chua

EXHIBIT A

J2EE Application Versioning Strategy

Sook Chuen Chua

[redacted]

Abstract

This paper presents a versioning mechanism for J2EE applications that ensures backward compatibility to support existing client applications that need flexibility in migrating to new versions. The method provides a controlled mechanism to support multiple application versions and allows dependent client applications the opportunities to migrate to new versions based on their own schedules and priorities. This is accomplished through methods that manage the introduction of new versions and controlled sun setting of older versions in a cost-effective fashion.

Introduction

A J2EE application is a multi-tiered, distributed application that commonly comprises of a web-tier that interacts with a business-tier and an enterprise information systems (EIS) tier. It is also possible for a J2EE application to consist only of business-tier and EIS-tier components; there is no web component. These J2EE applications are system-oriented where their clients are also applications systems. In order for inter-system communications to take place, application stubs must be distributed to the client systems.

As time progresses, enhancements and bug fixes are added to the application. These changes result in a new application version or release. With a web-based J2EE application, clients can be redirected to the new application version in an easy, seamless fashion with minimal system downtime through the use of hardware load balancers and additional hardware. For a system-oriented J2EE application, migrating clients to a new application version is more complicated as new version of the application stubs must be generated and distributed to client systems. The changes take effect only when the clients switch to using the new application stub. This "switching" action must be performed in a synchronous fashion between the client and the server applications. Otherwise, it results in binary incompatibility errors. This complexity is further compounded if the server application is a service provider that provides an array of services co-shared by multiple clients:

Clients X and Y share a service, *service A*, on machine *M1*. Client X requests for updates to *service A*. Deploying the new version of *service A* for client X on *M1* requires client Y to upgrade to avoid binary incompatibility errors. If the new *service A* does not work well and client X has to fall back to the old version, client Y also has to execute the fallback procedure. This "synchronous application upgrade" results in tight coupling between clients and in this case, causes unwarranted service discontinuity for client Y. To avoid tight coupling among clients, the service provider may deploy the new application version on machine *M2* and have client X point to *M2*, thus allowing client Y to continue to use the service on *M1*. This strategy, however, incurs additional hardware cost and effort to configure the new environment. The complexity, cost and risk increases with additional clients and increased frequency of application upgrades.

So, how should the server application manage multiple, concurrent application versions with minimal cost, effort, and risk? Also, how would the server application perform an application upgrade for each client in an isolated, precise fashion without service disruption to other clients?

This paper presents a versioning mechanism for J2EE applications that ensures backward compatibility to support existing client applications that need flexibility in migrating to new server side functionality and behaviour. The method provides a controlled mechanism to support multiple application versions and allow dependent client applications with opportunities to migrate to new versions based on their own schedules and priorities in a seamless and cost effective fashion. This is accomplished through methods that manage the introduction of new versions and controlled sun setting of older versions in a shared-resource environment with complete transparancy for the client applications.

Terms

"Application version" is used throughout this document to refer to application versions and versions.

Application Versioning Strategy

The JNDI naming service provides a list of public services, organized in a "tree" structure, that are hosted on an application server. A J2EE application server publicizes a service by binding each service into the JNDI tree with a name, referred to as the JNDI name. Clients then obtain a service by connecting to the application server and looking up the bound JNDI name of the service.

In a system-oriented J2EE application, services may be provided through Enterprise Java Beans (EJBs) and the latter will be bounded in the JNDI naming service. A remote client obtains an EJB service by connecting to the application servers and looking up the JNDI name of the specific EJB. With multiple application versions, it is extremely important to enforce unique JNDI names for these EJBs, internally within the application and also across all application versions. For example, clients *X* and *Y* use *serviceA* provided by *EJBA*. With 2 versions, this service will be bounded in the JNDI name service with names of *EJBA_R1* and *EJBA_R2* respectively. When client X switches from *serviceA* in version 1 to version 2, it will have to change from *EJBA_R1* to use *EJBA_R2* in version 2 as illustrated in the following diagram. This means the client has to be "version aware". How can this application version upgrade be performed for the client in a simple, transparent and controlled fashion? This can be achieved through the ***JNDI Proxy***.

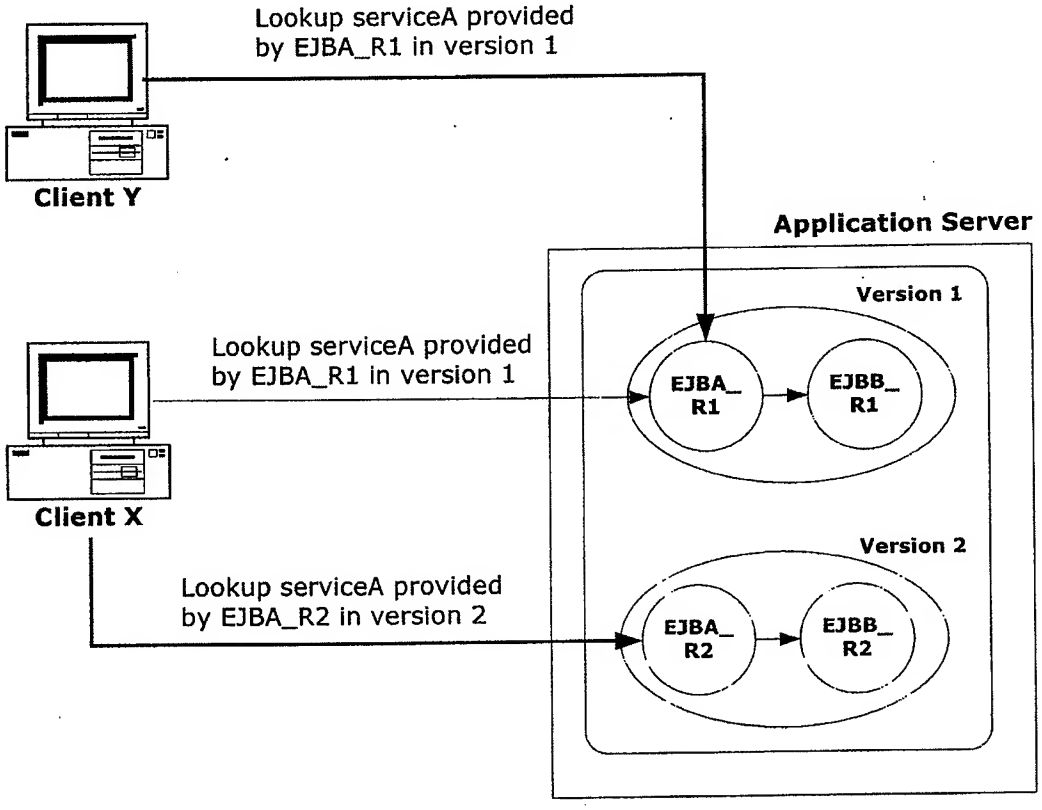


Figure 1: EJB Client looking up EJBs in different application versions

JNDI Proxy

Think of a machine *M* that sits behind a firewall. This machine is assigned with 2 IP addresses, a public IP address and a private IP address. The public IP address is publicized to the Internet and so all accesses to machine *M* are made through the public IP address. When a request for machine *M* reaches the firewall, the public address is translated to an internal, private IP address. This translation happens transparently, without client intervention. Using the same approach, the JNDI Proxy assigns 2 names to a service, a public ***service*** name and a private ***alias*** name. The ***service name*** is the public, official name of the service that is made known to clients. Any changes to this name require changes on the clients' end. As such, this name is not expected to change frequently. The ***alias name***, on the other hand, is a private name that is known only internally to the server application. It is used to bind a service to the JNDI naming context and is a unique value. Unlike the service name, this alias name changes with application versions.

The translation between the service name and the alias name is accomplished through a JNDI Proxy. This JNDI Proxy uses a combination of "Proxy" and "Singleton" design patterns to create a unique, single instance in the application that will proxy JNDI lookup requests from clients. So, when a client requires a service, it provides the service name which will be translated by the JNDI Proxy to the alias name. Details of the translation will be discussed in a later section. With this mechanism, the service name and alias name can change independently of application versions and the JNDI Proxy handles the change transparently for the client.

In addition to remote clients, the JNDI Proxy can also be used by internal application components to do lookup of other J2EE objects. For example, a remote client invokes *serviceA* provided by *EJB* that in turn invokes *EJBB*. If the client is directed to use *EJBA_R1* of version1, *EJBA_R1* will use the JNDI Proxy to perform JNDI lookup of *EJBB_R1* and not *EJBB_R2* in version 2. This mechanism allows developer to develop independently of application versioning changes. The following diagram illustrates the interactions among the remote client, external and internal EJB with the JNDI Proxy.

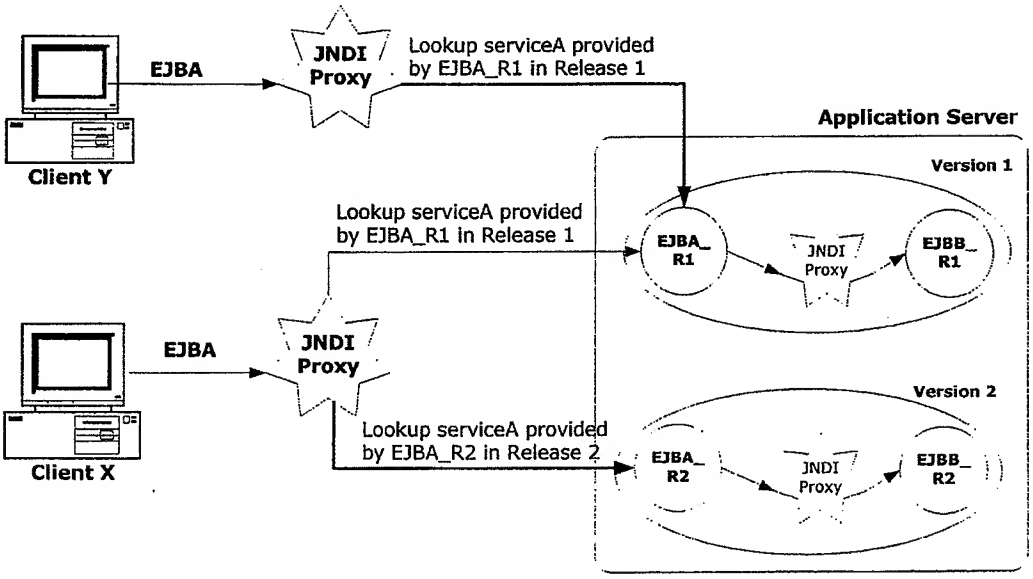


Figure 2: EJB client looking up EJBs through JNDI Proxy.

A situation may arise that an EJB of version 1 wants to invoke another EJB from a different version. For example, *EJBB_R1* in version 1 wants to invoke *EJBC_R2* in version 2 instead of *EJBC_R1* in version 1. The JNDI Proxy is capable of supporting this type of JNDI name translation.

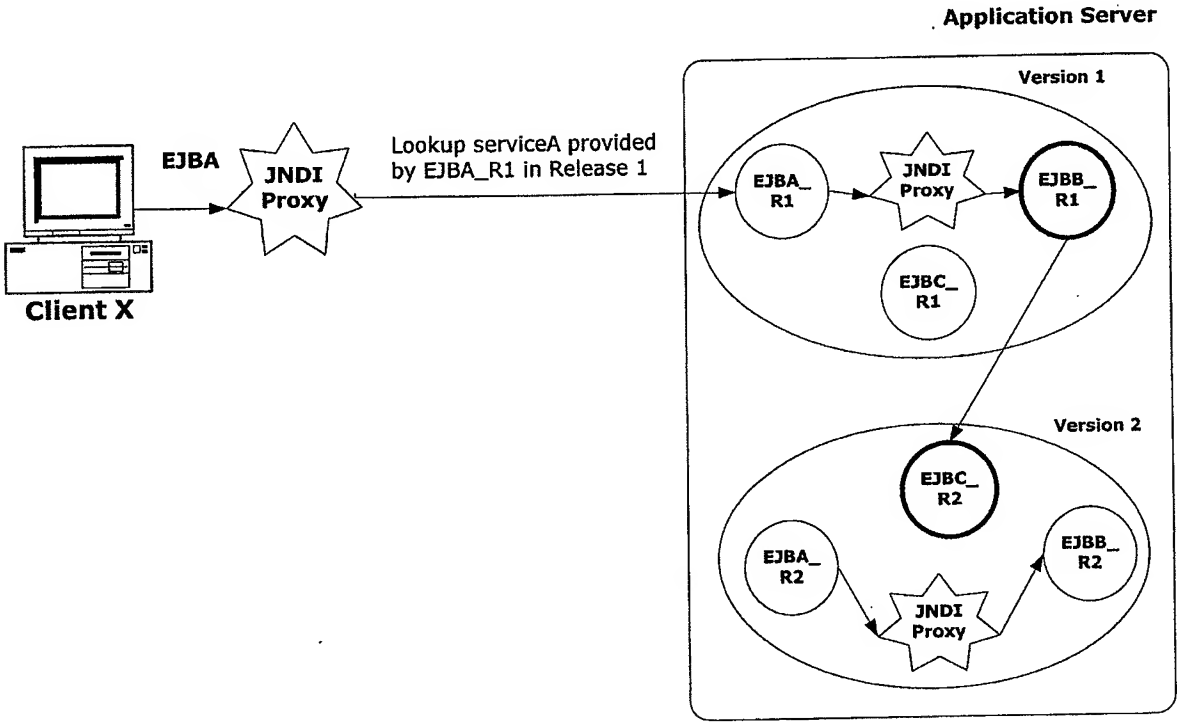


Figure 3: Indirect JNDI name translation by JNDI Proxy.

Benefits derived from JNDI Proxy

Using the JNDI Proxy to manage multiple, concurrent application versions yield the following benefits:

1. Allow application upgrade to be performed in an easy and transparent fashion for clients.
2. Provides client applications the flexibility to migrate to new application versions on their own schedules and priorities independently of other clients.
3. Establishes a strategy that the service provider can use to provide and manage multiple, concurrent versions of services for its clients in a simple, low risk and low cost manner.

Application versioning of J2EE Resources

The previous section describes how the JNDI Proxy supports application version management for services provided through EJB resources. There are additional J2EE resources that can be affected by application versioning. This section describes how the JNDI Proxy is used for application versioning management of these J2EE resources.

JMS resources

JMS resources include queues, queue connection factories, topics and topic connection factories. Like other J2EE resources, each JMS resource is bound by a name in the JNDI name service in the application server. These JMS resources can then be acquired and used by external clients or by internal EJBs.

In a scenario where there are multiple, concurrent application versions running, JMS resources may be referenced by resources in multiple application versions. As such, application versioning is required for these JMS resources. As pre-requisite to using the JNDI Proxy, each JMS queue must have a unique JNDI name. For example, a JMS queue, *QueueA*, in version 1 has a JNDI name of *QueueA_R1* and *QueueA_R2* in version 2. The following diagram illustrates how the JNDI Proxy plays a role in directing method calls from an EJB component to the JMS queue in 2 application versions. In version 1, *EJBB_R1* is directed by the JNDI Proxy to use *QueueA_R1* while in version 2, *EJBB_R2* is directed to use *QueueA_R2*.

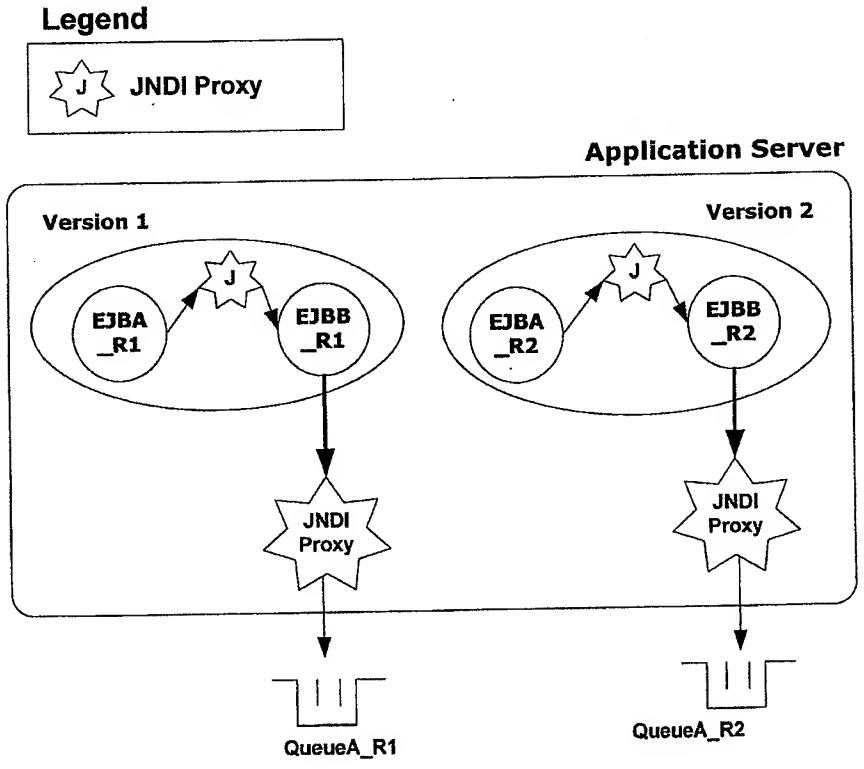


Figure 4: JMS queues with JNDI Proxy

In the above diagram, 2 queues are required and hence 2 JMS queue references are created. Alternatively, the 2 application versions may be able to reference the same JMS queue. In this situation, the JNDI Proxy can perform indirect JNDI name translation such that both *EJBB_R1* and *EJBB_R2* are directed to use *QueueA*.

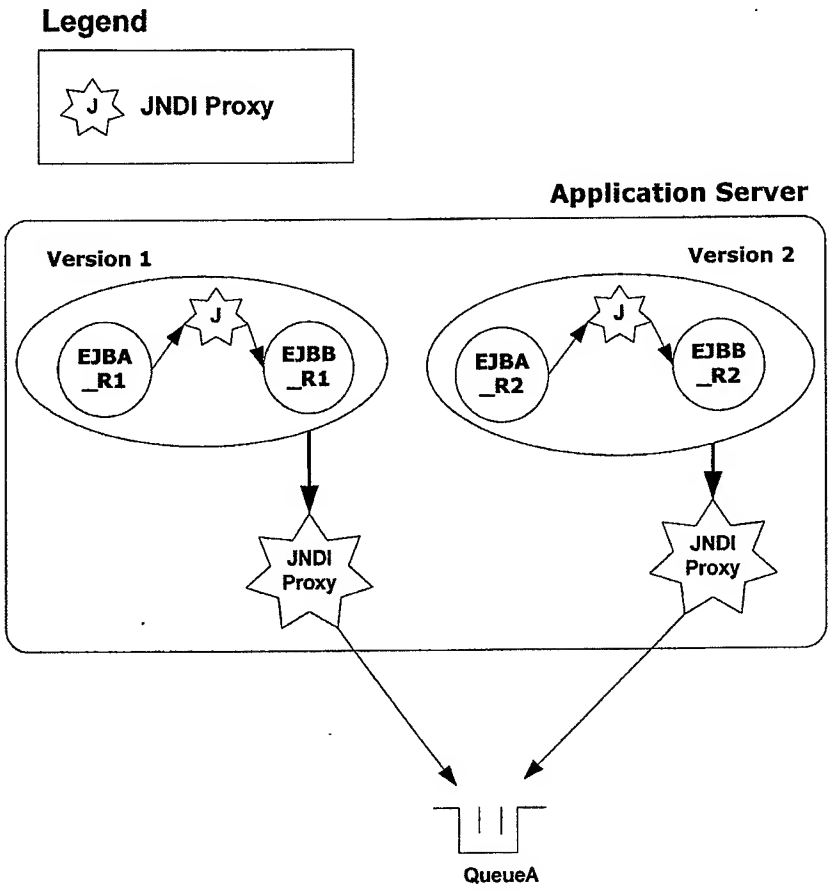


Figure 5: Indirect JMS queue name translation using JNDI Proxy

JDBC DataSource

The DataSource is a named group containing ready-to-use JDBC connections. It establishes database connections when the connection pool starts up and eliminates the overhead of establishing database connections on demand at runtime. A Datasource object enables JDBC clients to obtain a database connection. Each data source object has a unique JNDI name and points to a JDBC Connection Pool. A client, who can be a Java application, an EJB or a MDB, performs a JNDI lookup for the data source to get a database connection.

To support data sources for multiple application versions, configure different data source versions for different application versions. With this option, data sources will be created for each application version. Each data source will have a unique JNDI name, different from its peer in another application version. Application code that obtains database connection from data sources such as a Java client, an EJB or a MDB, will use the similar approach of going through the JNDI proxy to perform a transparent JNDI name translation to access the appropriate data sources intended for that version. The following diagram illustrates this configuration.

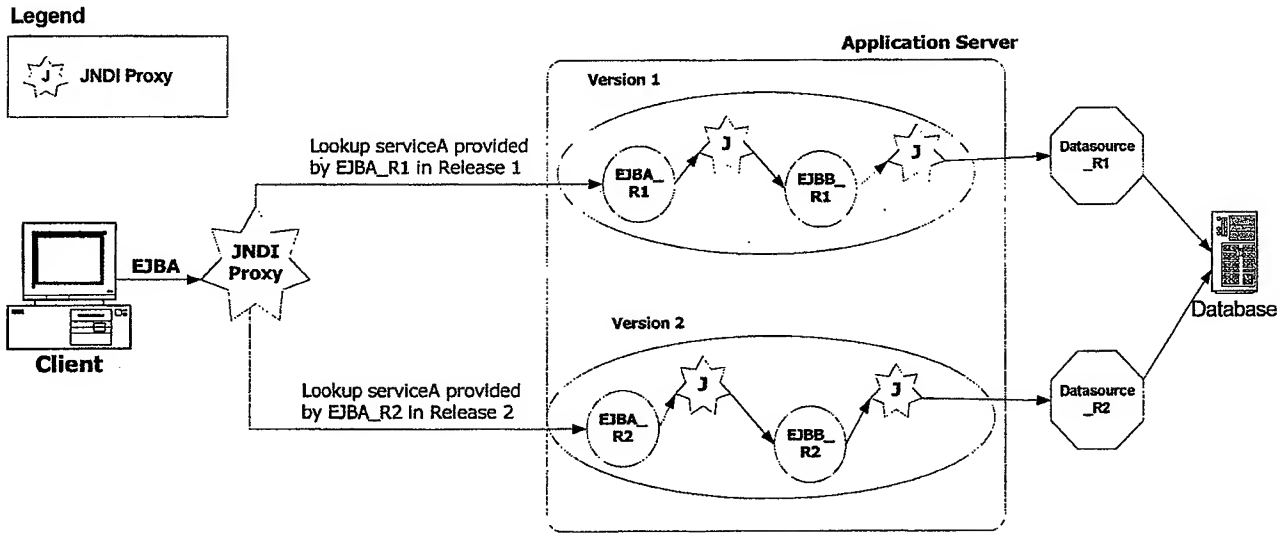


Figure 6: Multiple DataSources for multiple application versions

This approach is useful when applications have different requirements for the data sources between application versions. For example, version 2 uses *Datasource_R2* that has a maximum capacity of 100 connections while version 1 uses *Datasource_R1* that requires a maximum capacity of 200 connections.

If the situation does not require individual data source for each application version, the JNDI Proxy can use indirect JNDI name translation to allow multiple application versions to converge to a single data source as shown below. Note that with a single data source, any configuration changes to the data source may require restarting the application servers in order for the changes to take effect.

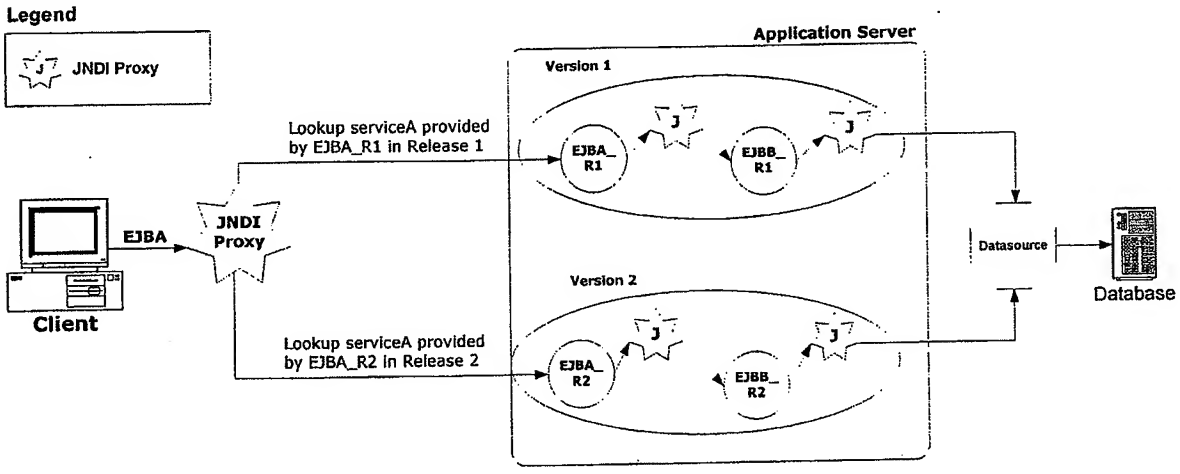


Figure 7: Single DataSource for multiple application versions

Design of JNDI Proxy

This section describes the detailed design of the components that collaborate to perform JNDI name translation in JNDI Proxy. The following diagram illustrates the class diagram of these components.

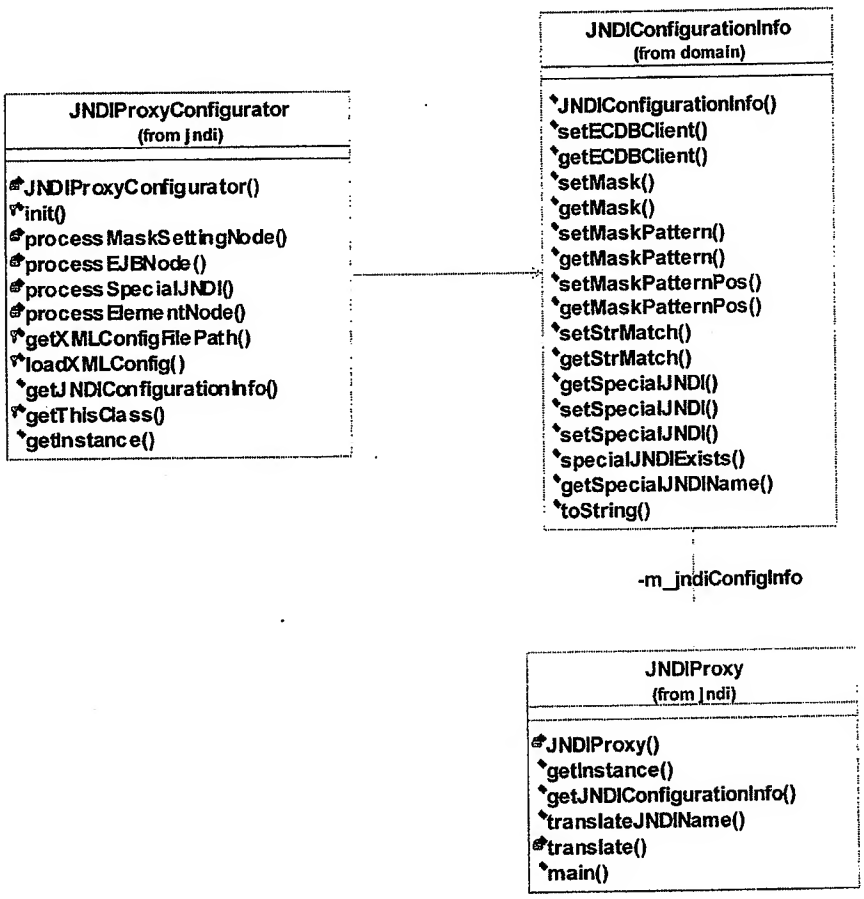


Figure 8: Class diagram

JNDIConfigurator

The JNDIConfigurator reads the JNDI configuration file that contains name translation instructions for the JNDIProxy to perform its name translations. The translation instructions in the configuration file are documented in an XML format that the JNDI Configurator understands.

Flowchart – Locating JNDI configuration file

The following diagram illustrates the process flow of the JNDIConfigurator in locating this configuration file:

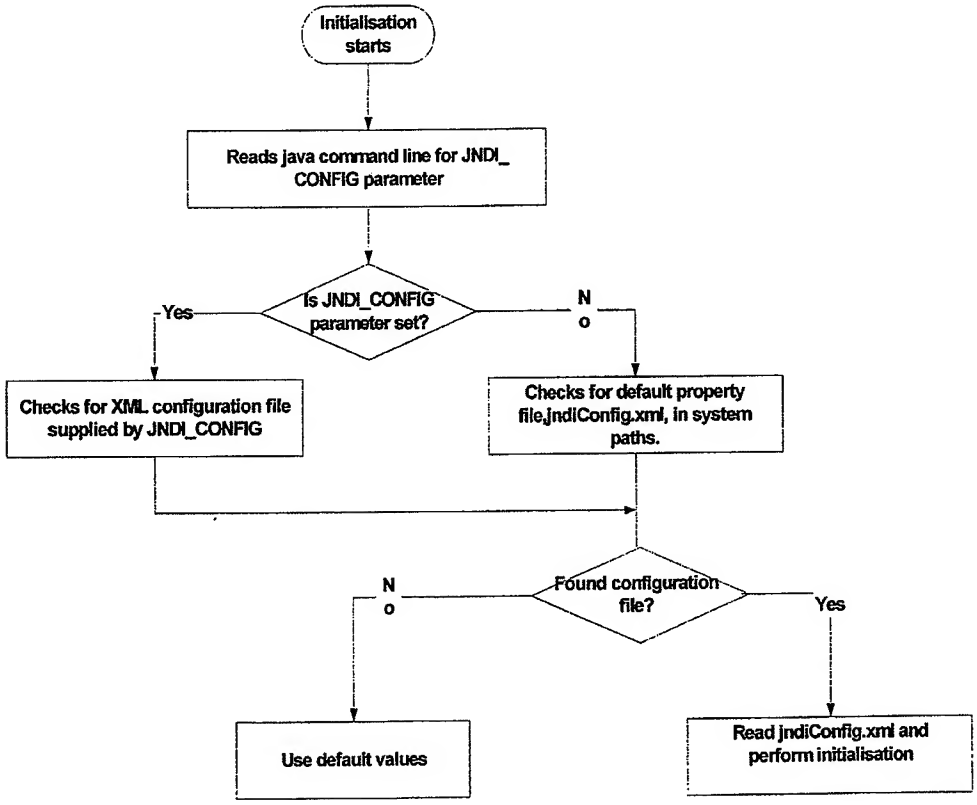
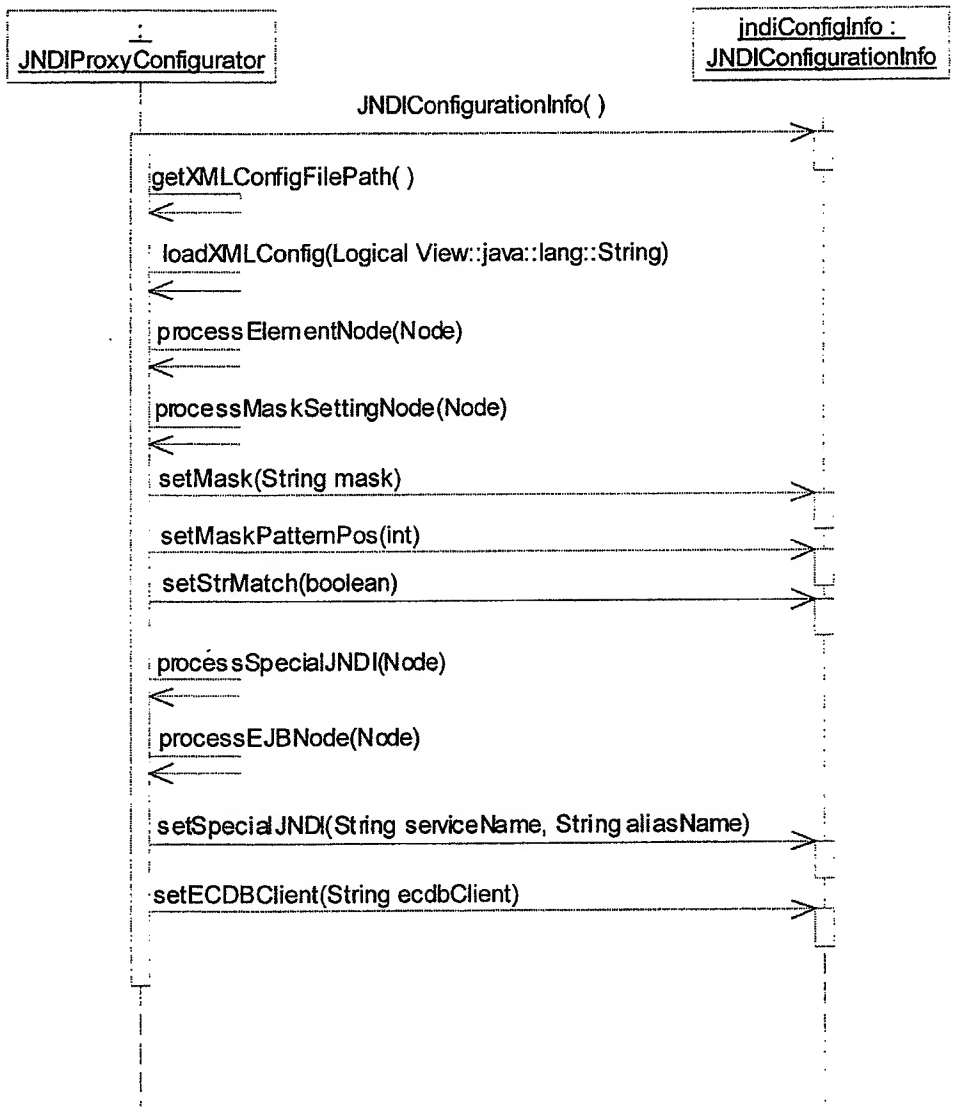


Figure 9: Locating JNDI configuration file

High Level Interaction Diagram

The following diagram illustrates how the JNDIConfigurator reads, parses the JNDI configuration file and loads the configuration information into a JNDIConfigurationInfo instance.



JNDI configuration file

The JNDI configuration file contains name translation instructions for the JNDIProxy in an XML format. The default configuration file is named ***jndiConfig.xml*** and is superceded by any property file provided through the Java system property **-DJNDI_CONFIG**. A sample configuration file and its XML elements are as follows:

```
<jndiConfig>
  <client>StockQuoteDisplay</client>
    <maskSetting>
      <mask>R212</mask>
      <maskPattern></maskPattern>
      <maskPos>0</maskPos>
    </maskSetting>
   <specialJNDI>
     <ejb>
       <serviceName>READONLY.Datasource</serviceName>
       <aliasName>READONLY.Datasource</aliasName>
     </ejb>
     <ejb>
      <serviceName>serviceQueryStockQuotes </serviceName>
      <aliasName>R213serviceQueryStockQuotes </aliasName>
     </ejb>
  </specialJNDI>
</jndiConfig>
```

Element	Description
jndiConfig	Root element of the XML file
client	Name of client that is using this property file.
maskSetting	Contains a group of mask setting values.
mask	Mask value is used to transpose the JNDI name. In this case, it is the application version or release number.
maskPattern	Specifies either name masking or position masking. When the MASK_PATTERN contains an integer value, the JNDIProxy will perform position masking on the JNDI names. So, a value of "0" means inserting the MASK-VALUE at the 0th position of the JNDI name, which in essence means a prepend operation. To append the MASK-VALUE to the JNDI name, specify a high value of "9999"; to interpose the JNDI name, specify a value between 1 and the length of the JNDI name.
maskPos	Represents the starting position in the JNDI name for inserting the mask value. The<maskPos> and <maskPattern> elements are mutually exclusive.
specialJNDI	Contains a group of name translation instructions to support "indirect" JNDI name translation illustrated in The JNDI Proxy will bypass the default direct JNDI name translation indicated in the <maskSetting> element.
ejb	Starting element for an EJB that requires "indirect" JNDI name translation.
serviceName	Public service name of the EJB that requires indirect JNDI name translation. This EJB is mapped to the alias name specified in the <aliasName> element. For example, "serviceQueryStockQuotes" will be mapped to "R213serviceQueryStockQuotes" instead of a direct name translation to "R212 serviceQueryStockQuotes".
aliasName	This is the internal alias name of an EJB in an indirect JNDI name translation.

JNDIProxy

The JNDIProxy component is the 'traffic cop" that re-directs clients to connect to their respective application version. This JNDI Proxy component is a singleton and proxies JNDI name lookup requests from clients. This section covers the detailed design of the JNDI Proxy component.

Sequence Diagram

The following diagram illustrates the interactions between a JNDIProxy client and the ***JNDIProxy.*** The JNDIProxy client may be an external client or an internal J2EE application component such as an EJB.

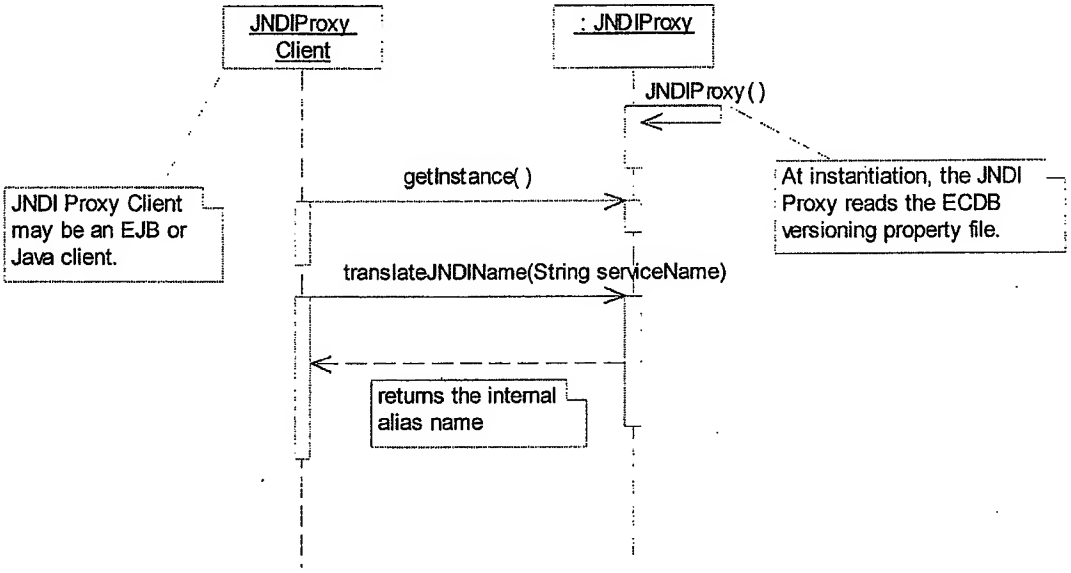


Figure 10: Sequence diagram illustrating a client's interactions with JNDIProxy

Sequence flow

1. The JNDIProxy is a singleton and has a private default constructor, ***JNDIProxy( ).*** At instantiation, the constructor reads a property file to initialize its parameters.

2. The JNDIProxy client invokes the ***JNDIProxy::getInstance( )*** to access the JNDIProxy instance.

3. The JNDIProxy client invokes the ***translateJNDIName( )*** method of JNDIProxy, passing in a component's service name and receiving its internal alias name.

JNDI Name Translation Logic

The following diagram illustrates the processes of JNDI name translation:

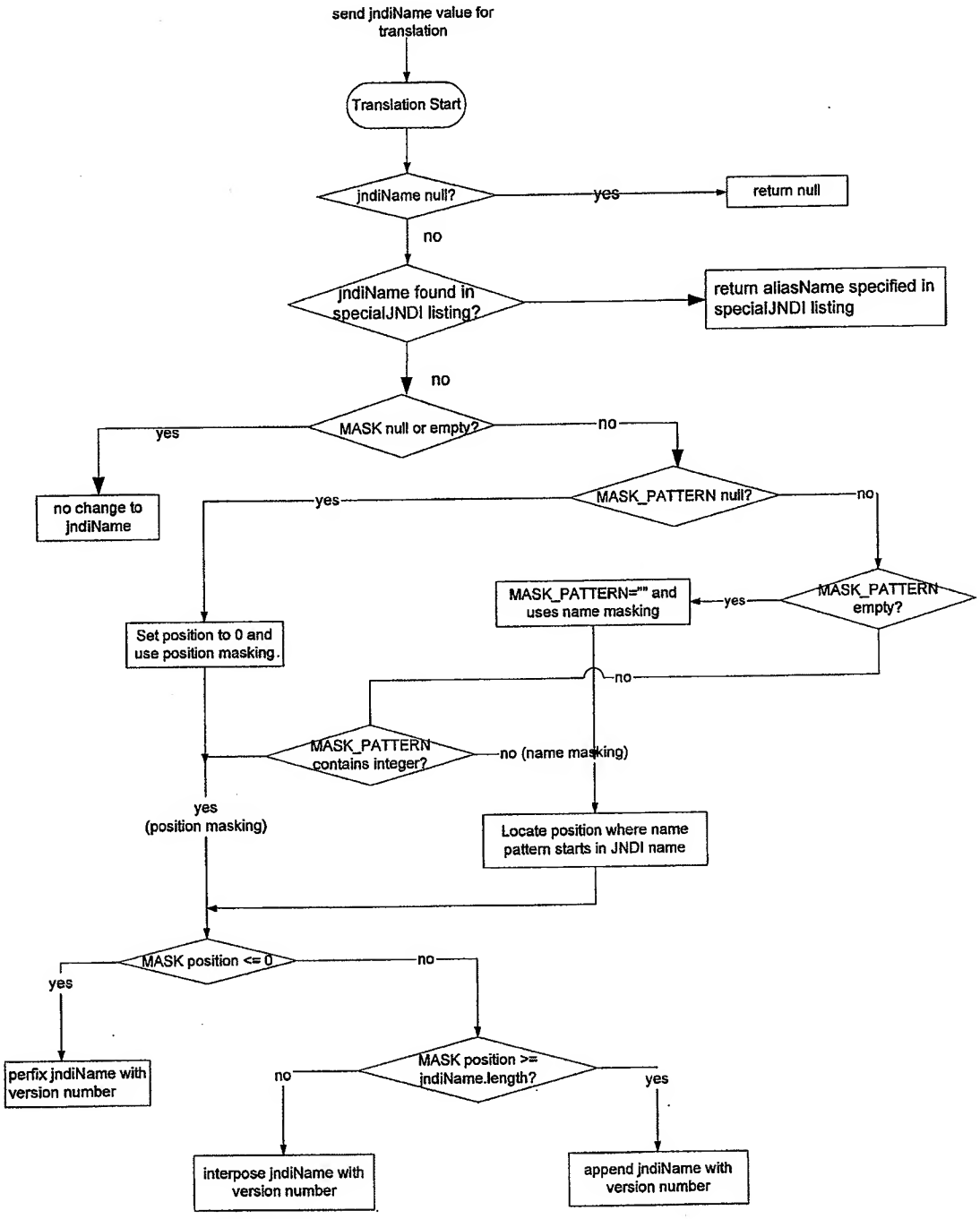


Figure 11: JNDI Name Translation

Sample JNDI Name Translation

Translation Example 1 :

Mask = R212
Mask Pattern = none
Mask Pos = 0

1. Input : serviceQueryStockQuotes
   Translated Name : R212 serviceQueryStockQuotes

2. Input : READONLY.Datasource
   Translated Name : READONLY.Datasource

Explanation :
When a mask position value is specified, the JNDIProxy performs position masking. In the above case, the mask position is 0 and this means that the mask value will be inserted as position 0. Hence, "serviceA" will be translated to "R212serviceA". However, "READONLY.Datasource" remains unchanged because the JNDI property file specifies that the translated JNDI name remains unchanged from the service name.

Translation Example 2 :

Mask = R212
Mask Pattern = service
Mask Pos = none

1. Input : serviceQueryStockQuotes
   Translated Name : R212QueryStockQuotes

2. Input : READONLY.Datasource
   Translated Name : READONLY.Datasource

Explanation :
When a mask pattern value is specified, the JNDIProxy performs pattern masking. In the above case, the mask pattern is "com" which means "service" will be replaced with "R212". Hence, "serviceQueryStockQuotes" is translated to "R212QueryStockQuotes". However, "READONLY.Datasource" remains unchanged because the JNDI property file specifies that the translated JNDI name remains unchanged from the service name.

Application Deployment

This section describes the procedures to prepare a J2EE application for deployment in a multi-application version environment. To faciliate discussion, a J2EE application named *ServiceApp* with the following package structure will be referenced :

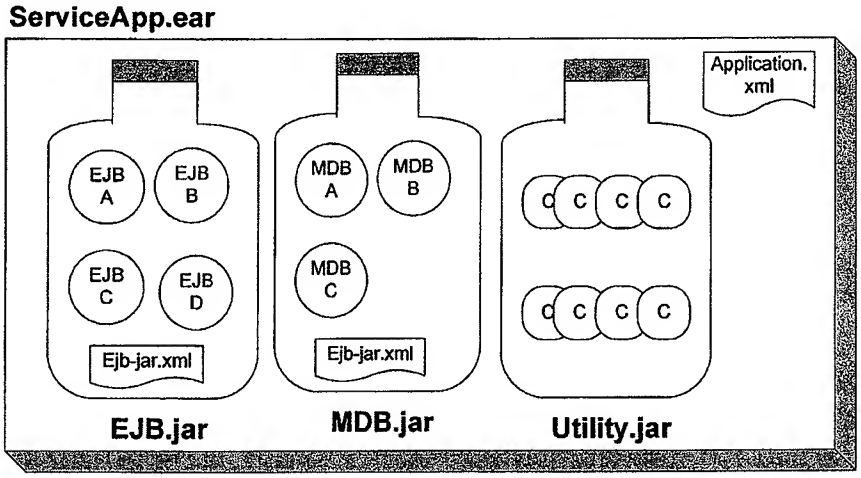


Figure 12 : A system-oriented J2EE application

This *ServiceApp.ear* contains 3 JARS, namely *EJB.jar, MDB.jar* and *Utility.jar*. The following table describes each of these modules:

J2EE module	J2EE Components	Deployment Descriptor
1. ServiceApp.ear	An EAR that contains one of more J2EE modules	application.xml
2. EJB.jar	An EJB jar that contains class files for session beans and an EJB deployment descriptor.	ejb-jar.xml
3. Utility.jar	A standard Jar that contains class files shared by one or EJB modules	Not applicable

The content of *application.xml* is as follows:

```
<?xml version="1.0"  encoding="UTF-8"?>
  <!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD
  J2EE Application 1.2//EN'
  'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>
  <application>
    <display-name>ServiceApp.ear</display-name>
    <description>A J2EE application</description>
    <module>
      <ejb>EJB.jar</ejb>
    </module>
  </application>
```

The *EJB.jar* contains 4 EJBs, namely *EJBA, EJBB, EJBC and EJBD*. This *EJB.jar* uses the Java classes in *Utility.jar* and the latter is referenced through the *MANIFEST.MF* in *EJB.jar* as follows:

```
Manifest-Version: 1.0
Created-By: 1.3.1_03 (Sun Microsystems Inc.)
Class-Path: Utility.jar
```

Managing deployment of multiple application versions

In most production environments, application servers are clustered to provide load balancing as well as high availability. With clustered objects, the application server may require homogeneous application deployment. This means the application is either deployed successfully to all application server instances or not deployed to any application server.

Consider an environment with a cluster of 4 application server instances. With a single application version, each application server instance in the cluster has a homogeneous copy of the J2EE application EAR module, *ServiceApp.ear*. To allow multiple versions of *ServiceApp.ear* to co-exist in a clustered environment, the J2EE application EAR modules must have unique names such as *ServiceApp_R1.ear* and *ServiceApp_R2.ear*. This naming convention provides unique identities and avoids overlaying. The diagram shows the co-existence of 2 versions of *ServiceApp.ear* deployed in a clustered environment:

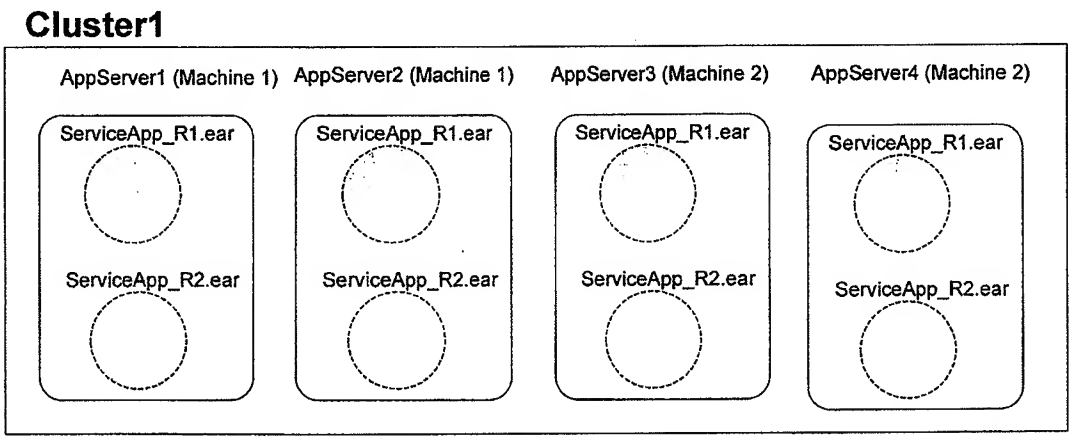


Figure 13: Co-existence of multiple J2EE EARs in the application server cluster.

In addition to unique names for the EARs, the J2EE modules within each EAR version also must have unique names. For example, *ServiceApp_R1.ear* contains *EJB_R1.jar* and *Utility_R1.jar* while *ServiceApp_R2.ear* contains *EJB_R2.jar* and *Utility_R2.jar*. Figure 14 shows the break down of these modules in each EAR version.

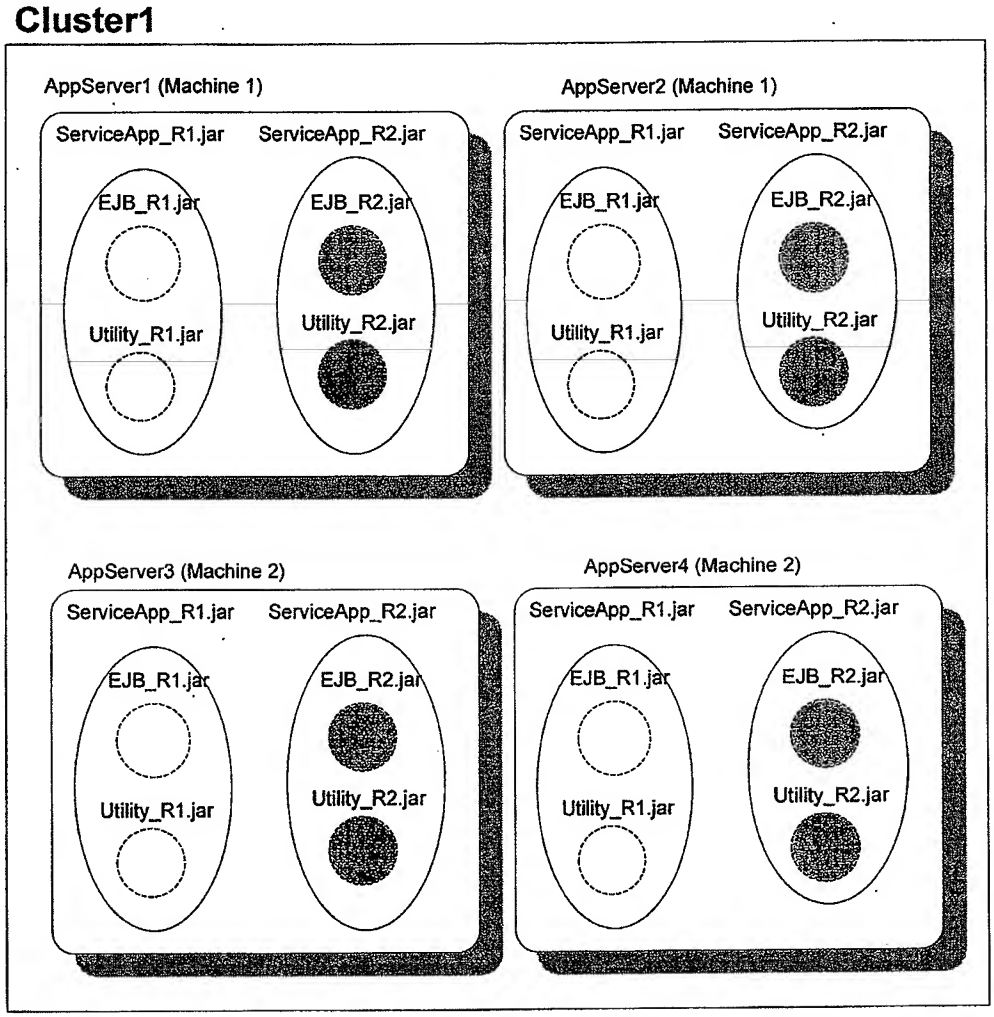


Figure 14: J2EE components in different EAR versions

Each J2EE EAR contains an *application.xml* file that describes the J2EE components packaged within it. With unique naming of J2EE modules, the content of *application.xml* in *ServiceApp_R1.ear* is as follows:

```
<?xml version="1.0"  encoding="UTF-8"?>
  <!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD
  J2EE Application 1.2//EN'
  'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>
  <application>
    <display-name>ServiceApp_R1.ear</display-name>
    <description>A J2EE application</description>
    <module>
      <ejb>EJB_R1.jar</ejb>
    </module>
  </application>
```

Similarly, the Utility JAR referenced by ***EJB_R1.jar*** in MANIFEST.MF is modified to the following:

```
Manifest-Version: 1.0
Created-By: 1.3.1_03 (Sun Microsystems Inc.)
Class-Path: Utility_R1.jar
```